pyHai Documentation

Release 0.1.2

Mark LaPerriere

January 19, 2012

CONTENTS

A system profiler/auditor written in Python that supports custom plugins and a custom profiler. Custom plugins can superscede builtin plugins to allow for a great deal of flexibility. Plugins follow naming conventions for ease of use.

CHAPTER

ONE

LICENSE

Copyright 2011 Mark LaPerriere

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER

TWO

PROJECT

http://bitbucket.org/marklap/pyhai

API Docs: html, tar.gz, zip, source

CHAPTER

THREE

BASIC USE

An example of running pyHai with all the default parameters and using the builtin auditing plugins.:

```
>>> from pyhai import pyhai
>>> pyhai.audit()
```

This will run all the builtin auditing plugins and return a dictionary object with detailed information about the local system. The dictionary keys are the lowercase names of the plugin files (minus the '.py' file name extension) and the contents are the results of the plugin's run.

CHAPTER

FOUR

ADVANCED USE

4.1 Running Additional, Custom Plugins

An example of running pyHai with a single additional plugin path. The builtin plugins will be run before the "custom" plugins found in the path supplied.:

```
>>> from pyhai import pyhai
>>> pyhai.audit('/path/to/my/plugins')
```

This will return the dictionary of results of the auditing run including builtin plugins and the custom plugins found in the path provided.

An example of running pyHai with a number of additional plugin paths.:

```
>>> from pyhai import pyhai
>>> pyhai.audit(['/path/to/my/plugins', '/path/to/some/other/plugins'])
```

This will return the dictionary of results of the auditing run including builtin plugins and all custom plugins.

Important: When using custom plugins, if you name your plugin the same as one of the builtin plugins, your plugin will superscede the builtin plugin. This allows users to provide their own plugins, but could cause confusion if it is unexcpected. For instance, if you write a plugin called 'environment', the builtin plugin - environment - will not run and your plugin will be run instead.

4.2 Suppressing Builtin Plugins

An example of running pyHai with a single custom plugin directory and without running the builtin plugins.:

```
>>> from pyhai import pyhai
>>> pyhai.audit('/path/to/my/plugins', enable_default_plugins=False)
```

Important: If you don't supply custom plugin path(s) and try to disable builtin plugins, an exception is raised - nothing to do.

CHAPTER

FIVE

EXTENDING

This section describes how to create your own auditing plugins and using a custom profiler.

5.1 Creating Auditing Plugins

5.1.1 Conventions

Plugin files must use all lowercase letters and seperate words with underscores.:

```
super_cool_widget.py
```

Important: Any files starting with an underscore (_) or having an extension other than .py are ignored. This can be helpful if you need some utility functions such as is present in the distributions plugins/linux/__init__.py module for various package manager support.

Plugin class names use CamelCase letters and reflect the plugin file name.:

```
class SuperCoolWidget(AuditorPlugin):
    pass
```

5.1.2 Extending

Plugin classes have a single required method: `run`. This method will be called as part of the default run.

We'll use the builtin "environment" plugin as an example. You can check out the source code in the builtin plugins for additional reference. Example Custom Plugin:

```
# file: environment.py

# import the auditing plugin base class
from pyhai import AuditorPlugin

# name your plugin the same as your file name.
class EnvironmentPlugin(AuditorPlugin):
    # provide a run method
    def run(self, *args, **kwargs):
        return dict(os.environ)
```

5.1.3 Hooks

There are two hooks provided in the auditing base class. If you provide them in your custom auditing class, pyHai will execute them as such.

The `before` method will be called before the `run` method is called. The results of the `before` method will passed along to the `run` method as a keyword argument: `before_results`.

The `after` method will be called after the `run` method is called. The results of the run are available in the plugin class by calling `self._get_results()`.

5.2 Creating a Custom Profiler

5.2.1 Extending

Simply import the profiler base class and extend it. There are three methods that need to be supplied by the custom profile class: `profile`, `system_class` and `system`. The profile method needs to return a dictionary with some basic information about the system running pyHai to help auditing plugins make desicions about how to audit.

A simple custom profiler:

```
from pyhai.profilers.base import ProfilerBase
import platform

class CustomProfiler(ProfilerBase):

    def profile(self):
        return {
            'env':'production',
            'system_class':platform.system().lower(),
            'system':platform.linux_distribution().strip().split()[0],
            }

    def system_class(self):
        return self.profile()['system_class']

    def system(self):
        return self.profile()['system']
```

5.2.2 Custom Profiler Functions

profile This should return a dictionary with some basic information about the local system.

The profile dictionary will be passed to each loaded plugin so it can be used to help plugins make runtime decisions.

system_class This should return a string representing the type of the local system.

The default profiler uses the builtin platform module and returns the results of str(platform.system().lower())

This relates to the plugins path as the second level diretory. For example, a `system_class` of **'linux'** will load plugins in the `plugins/linux` directory.

system This should return a string representing more specifically the type of local system.

The default profiler uses some very basic logic to determine the type of windows or linux host it's running on.

This relates to the plugins path as the third level directory. For example, a `system_class` of **'linux'** and a `system` of **'ubuntu'** will load plugins in the `plugins/linux/ubuntu` directory.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*